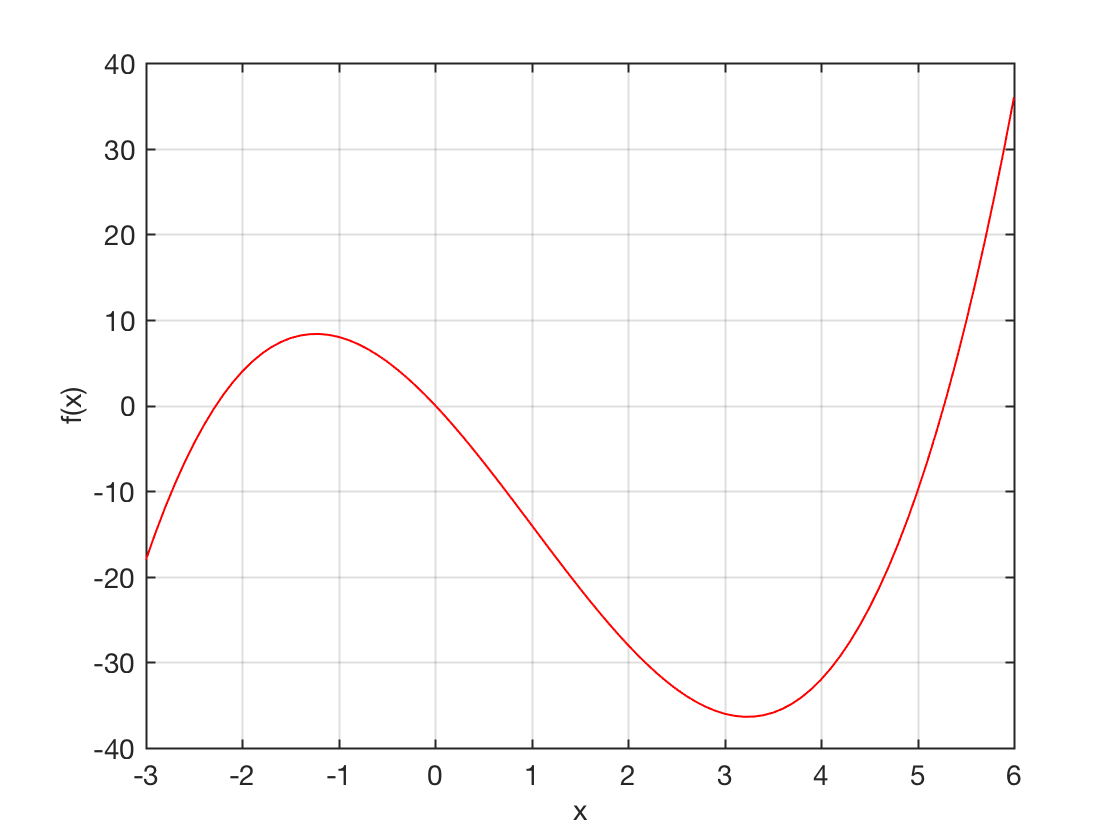
# **Unconstrained minimization.**

Here we are searching for the minimum (instead of the *zero*) of a function , where ***x*** are *n-independent variables*. There are two classes of algorithm competing for this goal, differing by whether they require the calculation of derivatives (1st only or 1st and 2nd) or not. We will start by considering the methods that go under the general definition of *conjugate gradients*. These methods use different techniques for constructing *conjugate directions*. The so-called *zero-order methods* work with only; the *first-order methods* utilize both and ; the *second-order methods* utilize , , and . The first-order and second-order methods are computationally more efficient, but are possible only when or both and are available.

**Zero order methods.**

**Line search.** A *line search* is a method to find the minimum of a function starting from a point and moving along a line parallel to a given *search vector*, , within some lower and upper bounds, *lb* and u*b***;** the method finds the next iterate of the form:

where denotes the current iterate, is the search direction, and is a scalar parameter.

How do we find the minimum of a function *f(x)* of a single variable *x*, with the constraints ? A plot of such a function could show a *global* minimum in between the boundaries and a *local* minimum at one of theboundaries. For example, consider the function, for :

clear, clc, close all

f = @(x) x.^3 - 3\*x.^2 - 12\*x

xvec = [-3:0.1:6]';yvec = f(xvec);

plot(xvec,yvec,'-r'); xlabel('x');ylabel('f(x)');grid on;hold on

In this case finding the global minimum is straightforward as all *stationary points* can be located by finding the roots of , and each constraint boundary can be checked for a global minimum by evaluating *f(lb)* and *f(ub)*. However, the function *f* may be difficult or impossible to differentiate analytically, as it would be the case for a complex algorithm.

The line search method attempts to decrease the objective function along the line in two steps:

1. The *bracketing phase* determines the range of points on the line to be searched. The bracket corresponds to an interval, typically {*a* *b*}, specifying the range of values of . We start with an initial value *f(x0)* and we calculate additional values *f(x1)*, *f(x2)*, *f(x3)*, until we reach the point *f(xn)*, where *f(x)* increases for the first time. In this process, it is a good idea to use a step that increase constantly (i.e., , with *c =* 1.6180033989… = 1 + *R*, where *R* is the *Golden ratio*, see below) in order to reach the minimum quickly, even if the resulting bracket is wide. In the end, the minimum point is bracketed in the interval (*xn-2*, *xn*).

2. The *sectioning phase* narrows the subinterval progressively to identify the function minimum within an error tolerance. Suppose the minimum has been bracketed in the interval {*a b*} of length *h.* We first evaluate the function at:

, .

If the minimum lies in the interval:

otherwise in:

Assuming that we set:

which yields a new interval:

We only need to evaluate:

and we are ready for the next iteration.

With respect to the value of , it’s important to notice that the distance:

in the top diagram is the same as the distance:

in the bottom diagram, leading to:

substituting we obtain:

(

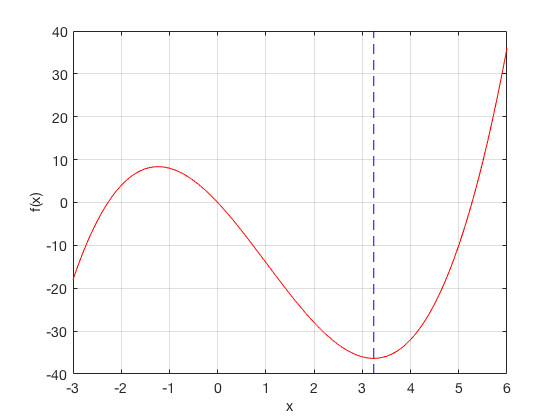
(

which can easily be solved for :

p = [-1 -1 1];

r = roots(p)

leading to a positive solution , which is the value of the *Golden Ratio*. This number was known to the ancient Greeks as the ratio of the sides of the *golden rectangle*, the rectangle with the most pleasing proportions.



This *line search* algorithm is implemented in the function *line\_search*.  We are now ready to complete the identification of the minimum for :

x = 2;

lb = -3;

ub = 6;

[x\_min,f\_min] = line\_search(f,x,-3,6)

x\_min = 3.2361

f\_min = -36.3607

vline(x\_min,'b--')

**Conjugate Directions (Powell method).** First, we recall that the *Taylor series expansion* of a function of a single variable about the point is the infiniteseries:

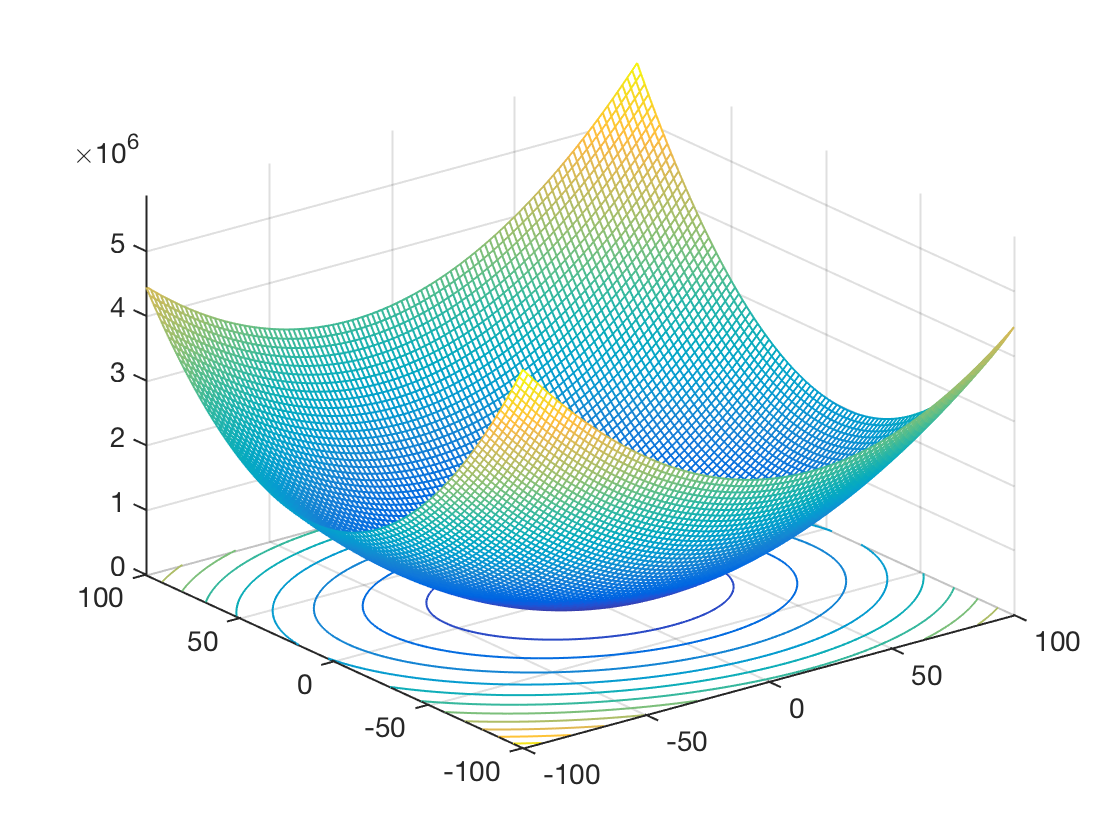
This series is meaningful only if all the derivatives of exist at and the series converges. In general, convergence occurs only if is sufficiently close to , that is if:

where is the *radius of convergence* of the series.

In the *Powell* method we apply this expansion to a function of several variables, and is now a vector. Regardless of the form of we can find its *quadratic approximation* in the neighborhood of by truncating the *Taylor series expansion* of about .

where is the *gradient* (vector of 1st derivatives) of at , and is the *Hessian* (matrix of 2nd derivatives) of at . Taking leads to the quadratic approximation of our original function , which we can write as:

which is the general matrix representation of a *quadratic surface.*

**

*b = [5 3]';*

*H = [25 2;1 20];*

*H = H'\*H;*

*eig(H)*

*x = [-100:2:100;-100:2:100];*

*[m,n] = size(x);*

*func = @(x) 10 + b'\*x +1/2\*x'\*H\*x*

*for i = 1:n*

*for j = 1:n*

*f(i,j) = func([x(1,i);x(2,j)]);*

*end*

*end*

*[X,Y] = meshgrid(x(1,:),x(2,:));*

*meshc(X,Y,f)*

Differentiating with respect to we obtain:

where again is the gradient of at . Given an initial point , we can carry out a *line search* in the direction of a unit vector and move by a distance *s* from to a new point , such that the value of is the smallest possible along that direction. As a result,the gradient changes by :

If is perpendicular to a unit vector such that:

we say that and are mutually *conjugate* or ***H*** *normal* to each other. The implication is that once we have minimized in the direction of we can move along without ruining the previous minimization.

Start with point ***x0***

**⇓**

Choose vectors ***v1*** and ***v2***. The usual choices

are unit vectors in the ***x1*** and ***x2*** directions.

**⇓**

Find the distance *s*1 that minimizes *f*(***x0*** + *s1****v1***).

The new point is ***x1*** = ***x0*** + *s*1***v1***

**⇓**

Find the distance *s*2 that minimizes *f*(***x1*** + *s2****v2***).

The new point is ***x2*** = ***x1*** + *s*2***v2***

**⇓**

The last search direction is ***v3*** = ***x2*** − ***x0***.

**⇓**

Find the distance *s*3 that minimizes *f*(***x0*** + *s3****v3***).

The new point is ***x3*** = ***x0*** + *s*3***v3***

**⇓**

Choose new vectors ***v1*** and ***v2***:

a. Discard old ***v1***

b. ***v2*** becomes ***v1***.

c. ***v3*** becomes ***v2***.

d. ***x3*** becomes ***x0***



For a perfectly quadratic function of *n-independent* variables it is possible to construct *n-conjugate* directions, and it would take *n-cycles of* *n+1* *line searches* to reach the minimum.

The following is how the *Powell method* works to find the minimum on a 2-dimensional surface.

The described cycle is repeated twice: the figure shows the two cycles superimposed on the contour map of a quadratic surface. The first cycle starts at point ***x***0 and ends at ***x***3. The second cycle brings to the minimum ***x***6. The directions ***x***0***x***3 and ***x***3***x***6 are *mutually conjugate*.

In practice, in order to obtain an efficient progression towards the minimum, experience has shown that, rather than discarding at the end of each cycle, it is better to discard the direction that resulted in the *largest decrease* of , as this is likely to be close to the 1st direction added in the next cycle. As a result of the change, the search directions are no longer strictly *conjugate*, and finding the minimum of usuallytakes more than *n* cycles.

It is important to realize that the Powell method *does not require the calculation of derivatives*: a function based on this method is shown below. It calls the function *line\_search* (see above) to find the minimum in each line search:

function [x\_min,f\_min,niter] = powell(f,x,h,tol,maxiter)

% f = handle to the function that returns f(x).

% x = starting point as a column vector.

% h = initial increment for the line search.

% tol = error tolerance on x.

if nargin < 5; maxiter = 1000; end

if nargin < 4; tol = 1.0e-8; end

if nargin < 3; h = 0.05; end

ndim = length(x);       % Number of dimensions

V = eye(ndim);          % Matrix containing the unit vectors in the search directions

dcr\_f = zeros(ndim,1);  % Vector storing the decreases in f(x)

x\_old = 0;

niter = 0;

while norm((x-x\_old)/ndim) > tol

    if niter >= maxiter

        disp('Warning: Powell did not converge in the maximum ');

        disp('number of iterations. Result may be inaccurate.');

        return

    end

    niter = niter + 1;

    x\_old = x;

    f\_old = f(x\_old);

    % Here we do a line search for each dimension.

    for n = 1:ndim

        v = V(:,n);

        % In the line search we scan on the multiplier *s* of the unit vector

        % in the search direction.

        [s,f\_min] = line\_search(@(s) f(x+s\*v),0.0,-Inf,Inf,h);

        dcr\_f(n) = f\_old - f\_min;

        f\_old = f\_min;

        x = x + s\*v;

    end

    % A final line search completes the iteration.

    v = x - x\_old;

    [s,f\_min] = line\_search(@(s) f(x+s\*v),0.0,-Inf,Inf,h);

    x = x + s\*v;

    % Here we update the search directions discarding the direction that

    % produces the biggest decrease in f(x).

    [~,max\_ind] = max(dcr\_f);

    for n = max\_ind:ndim-1

        V(:,n) = V(:,n+1);

    end

    V(:,ndim) = v;

    x\_min = x;

end

end

Here we use *powell* to find the minimum of a quadratic function in 2 dimensions:

func = @(x) 10\*x(1)^2 + 3\*x(2)^2 -10\*x(1)\*x(2) + 2\*x(1)

x = [1 1]’

[xmin,fmin,niter] = powell(func,x,0.01)

xmin = [-0.6000 -1.0000]

fmin = -0.6000

niter = 5

**First order methods.**

**Conjugate Gradients (CG methods).** We start again with a function that has a quadratic form. Given a direction **v**, with the Powell’s method *n* line minimizations are required to construct a single conjugate direction. First order methods can reduce the identification of each conjugate direction to a single line minimization.

We already introduced the *Conjugate Gradients*, CG, methods when we discussed the *iterative* solution of a system of linear equations:

where ***A*** (*m x n*) is *symmetric* and *positive definite*. We noticed how solving this system is the same as minimizing:

In fact, differentiating with respect towe obtain the *gradient* of :

and is at its minimum when (*First Order Optimality Condition*, FOC).

The CG method is guaranteed to converge to the solution:

in at most *n* iteration.The method can be particularly useful if the matrix is *sparse* (for example, left panel below), and thus there is no need to calculate its inverse , which is very dense (right panel below).

We start with an initial vector and compute a refined solution at each cycle:

⇓

⇓

⇓ ... and so on

The *step length* is chosen so that minimizes in the *search direction* . This means that must *tend to* satisfy:

Introducing the *residual*:

and pre-multiplying both sides by we find :

We notice here that the expression derived for is the general optimizer for the selection of a step length in *line search* methods in which a quadratic function is minimized along the line **.**

The obvious choice for the search direction would be the *steepest descent vector* (the negative of the gradient):

However, while providing the correct result, this method converges very slowly (convergence order *p = 1,* see below), because consecutive searches tend to be in approximately the same direction. To avoid this problem, the *conjugate gradient* method uses a modified search direction:

The constant is chosen so that the two successive search directions are *not interfering* with each other; these directions are *conjugate* or ***A*** *normal* because:

Substituting for we obtain:

The following is a flowchart of the *conjugate gradients* algorithm (function *conj\_grad*):

Start with any

**⇓**

Use the *steepest descent* as 1st step

**⇓**

**⇓**

Loop with *k = 1 to n* where *n*

is the number of variables*.*

**⇓**

If exit loop (convergence criterion = error tolerance is met)

The scheme can be further modified by noticing that:

since , . Moreover, given that we have:

despite the fact that .

Start with any

**⇓**

Use the *steepest descent* as 1st step

**⇓**

**⇓**

Loop with *k = 1 to n* where *n*

is the number of variables*.*

**⇓**

If exit loop (convergence criterion = error tolerance is met)

This modified scheme is implemented in the function *conj\_grad\_mod*:

function [ x,r,i ] = conj\_grad\_mod(A,x,b,tol)

if nargin<4

tol = 1e-9;

end

n = length(b);

r = b - A\*x; s = r;

i = 0;

rtr = r'\*r;

while norm(r) > tol

i = i+1;

As = A\*s;

alpha = rtr/(s'\*As);

x = x + alpha\*s;

r = r - alpha\*As;

rntrn = r'\*r;

beta = rntrn/rtr;

s = r + beta\*s;

rtr = rntrn;

end

end

A = [2 3 1;3 1 1;1 1 2]; A = A'\*A;

b = [3 2 1]';x0 = [0 0 0]'

[x,r,i] = conj\_grad\_mod(A,x0,b)

The modified scheme provides the core of the **Fletcher–Reeves (FR) method** for finding the minimum of a quadratic function in *n* iterations. Since and ***b*** are not known, the method requires the calculation of the *gradient* (thereby the classification as a 1st order method) of the function at each new point along the minimization. A flowchart of the algorithm is shown below:

Start with point

**⇓**

Use the *steepest descent* as 1st step

**⇓**

**⇓**

Loop with *k = 1 to n* where *n*

is the number of variables*.*

**⇓**

Do a Line Search to find

If or | exit loop

(error tolerance is met)

FR gets its favorable convergence properties from the conjugacy of the search directions near the optimum. If we start “far” from the optimum, the algorithm does not necessarily gain anything from maintaining this conjugacy. The *n*-step convergence is only guaranteed when we start with a steepest-descent step, and the model is quadratic. Therefore, we can periodically restart the algorithm, by taking a steepest-descent step from the last identified minimum.

function [x\_min,f\_min,niter] = fletcher\_reeves(f,x0,h,tolgrad,tolfun,tolx,nder)

% f = function to be minimized

% x0 = starting point.

% h = initial search increment.

% nder = type of derivative: [1] 1st forward [2] 2nd forward [3] central

% tolgrad = error tolerance on the gradient (default = 1.0e-4).

% tolfun = error tolerance on the function (default = 1.0e-6).

% tolx = error tolerance on the solution (default = 1.0e-6)

% niter = number of iterations

if nargin < 7; nder = 2; end

if nargin < 6; tolx = 1.0e-6; end

if nargin < 5; tolfun = 1.0e-6; end

if nargin < 4; tolgrad = 1.0e-4; end

if nargin < 3; h = 0.1; end

[f0,g0] = gradx(f,x0,nder);

r0 = -g0;

s0 = r0;

df = tolfun;dx = tolx;

niter = 0;

while (r0'\*r0) >= tolgrad || abs(df) >= tolfun || (dx'\*dx) >= tolx

        niter = niter + 1;

        [alpha,~] = line\_search(@(alpha) f(x0+alpha\*s0),0.0,-Inf,Inf,h);

        x\_min = x0 + alpha\*s0;

        [f\_min,g1] = gradx(f,x\_min,nder);

        r1 = -g1;

        beta = (r1'\*r1)/(r0'\*r0);

        s1 = r1 + beta\*s0;

        dx = x\_min-x0;df = f\_min-f0;

        x0 = x\_min;f0 = f\_min;s0 = s1;r0 = r1;

end

end

Here is an example of a *Fletcher-Reeves* minimization with a quadratic function:

func = @(x) 10\*x(1)^2 + 3\*x(2)^2 -10\*x(1)\*x(2) + 2\*x(1)

x = [1 1]'

[x\_min,f\_min,niter] = fletcher\_reeves(func,x,0.01)

x\_min = [-0.6000 -1.0000]

f\_min = -0.6000

niter = 12

**Second order methods.**

Another *Taylor series* is the *expansion* of a function of a single variable about an arbitrary value of is:

The effect of truncating the series is of great practical importance. For example, keeping the first *n*+1 terms, we have:

The *truncation error* (sum of the truncated terms starting with the term including ) is:

which is a concise way of saying that is of the *order* of, or behaves as .

In thesecond order methods we apply this expansion to a function of several variables, and is now a vector. Regardless of the form of we can find the *quadratic approximation* of its *Taylor expansion* about as:

where again the vector is the *gradient* of and (or ) is the *Hessian* of .

In particular, we can consider as a unit vector in the direction of a possible *line search* that we intend to pursue to find the minimum of . The intuitive choice for is the *steepest descent vector*:

However, if the function is smooth enough, and we know the Hessian, we can select to be the so called ‘*Newton*’ direction. Since the minimum of is achieved when its 1st derivative is 0, we differentiate the RHS of the Taylor expansion with respect to and then set the result to zero:

As long as the Hessian is *positive definite* (and the *rcond* number >10E-14) the direction of search, , is always in a descent direction of . In fact, multiplying by we obtain:

If is *positive definite* so is , and , which means it decreases the magnitude of .

We notice here that the *Newton* direction is more expensive that the *steepest descent* direction because we must compute the Hessian and invert it. However, the order *p* of convergence (not to be confused with the search direction from above) of the steepest descent methods is linear (*p=1*). This is because, as we approach the minimum of , the value of changes according to:

(where *k* is the iterationnumber and is the point where attains its minimum), while the *order* of convergence of the Newton methods is quadratic (*p=2*):

In these limits must be a *constant* (thus, neither 0 nor ), and if this constant exists it is called the *rate of convergence* (or more generally, the *asymptotic error*); *p* is the *order of convergence*. The following are some numerical examples.

Let . This is a case of *sublinear convergence*:

In this case, the rate of convergence is *r = 1* and the order of convergence is *p = 1*.

clear x Q1 Q2 Q3

niter = 1000;

for k = 1:niter

x(k) = 1/k;

end

Q1 = abs(x(2:end))./(abs(x(1:end-1)).^0.5);

Q2 = abs(x(2:end))./(abs(x(1:end-1)).^1);

Q3 = abs(x(2:end))./(abs(x(1:end-1)).^2);

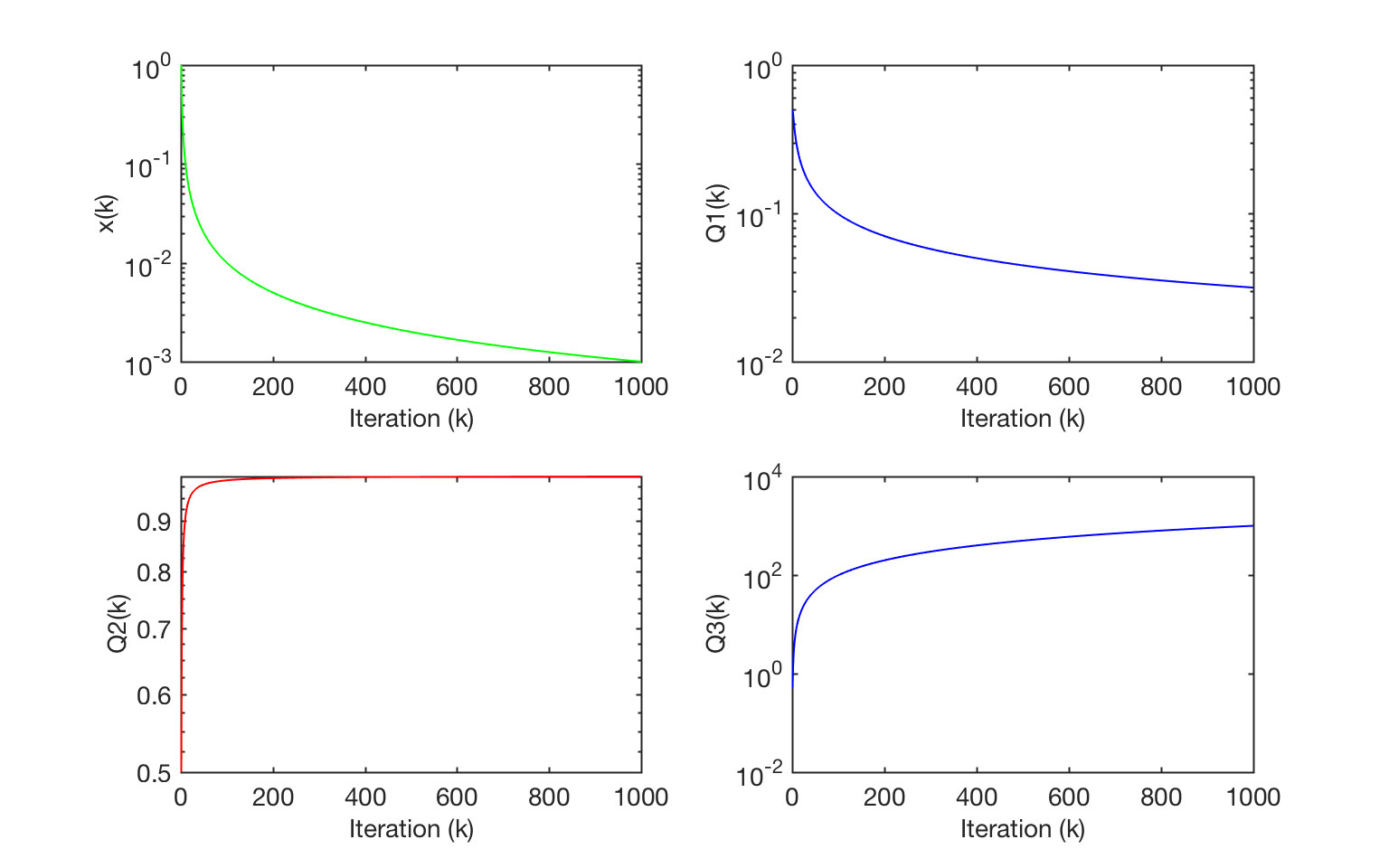
SubLinear\_convergence = figure;set(gcf,'Unit','Normalized','Position',[0 0.4 0.6 0.6])

subplot(2,2,1);semilogy([1:niter],x,'-g');xlabel('Iteration (k)');ylabel('x(k)')

subplot(2,2,2);semilogy([1:niter-1],Q1(1:niter-1),'-b');xlabel('Iteration (k)');ylabel('Q1(k)')

subplot(2,2,3);semilogy([1:niter-1],Q2(1:niter-1),'-r');xlabel('Iteration (k)');ylabel('Q2(k)')

subplot(2,2,4);semilogy([1:niter-1],Q3(1:niter-1),'-b');xlabel('Iteration (k)');ylabel('Q3(k)')



Let . This is a case of *linear convergence*:

In this case, the rate of convergence is *r = 1/2* and the order of convergence is *p = 1*.

clear x Q1 Q2 Q3

niter = 1000;

for k = 1:niter

x(k) = 1/(2^k);

end

Q1 = abs(x(2:end))./(abs(x(1:end-1)).^0.5)

Q2 = abs(x(2:end))./(abs(x(1:end-1)).^1)

Q3 = abs(x(2:end))./(abs(x(1:end-1)).^2)

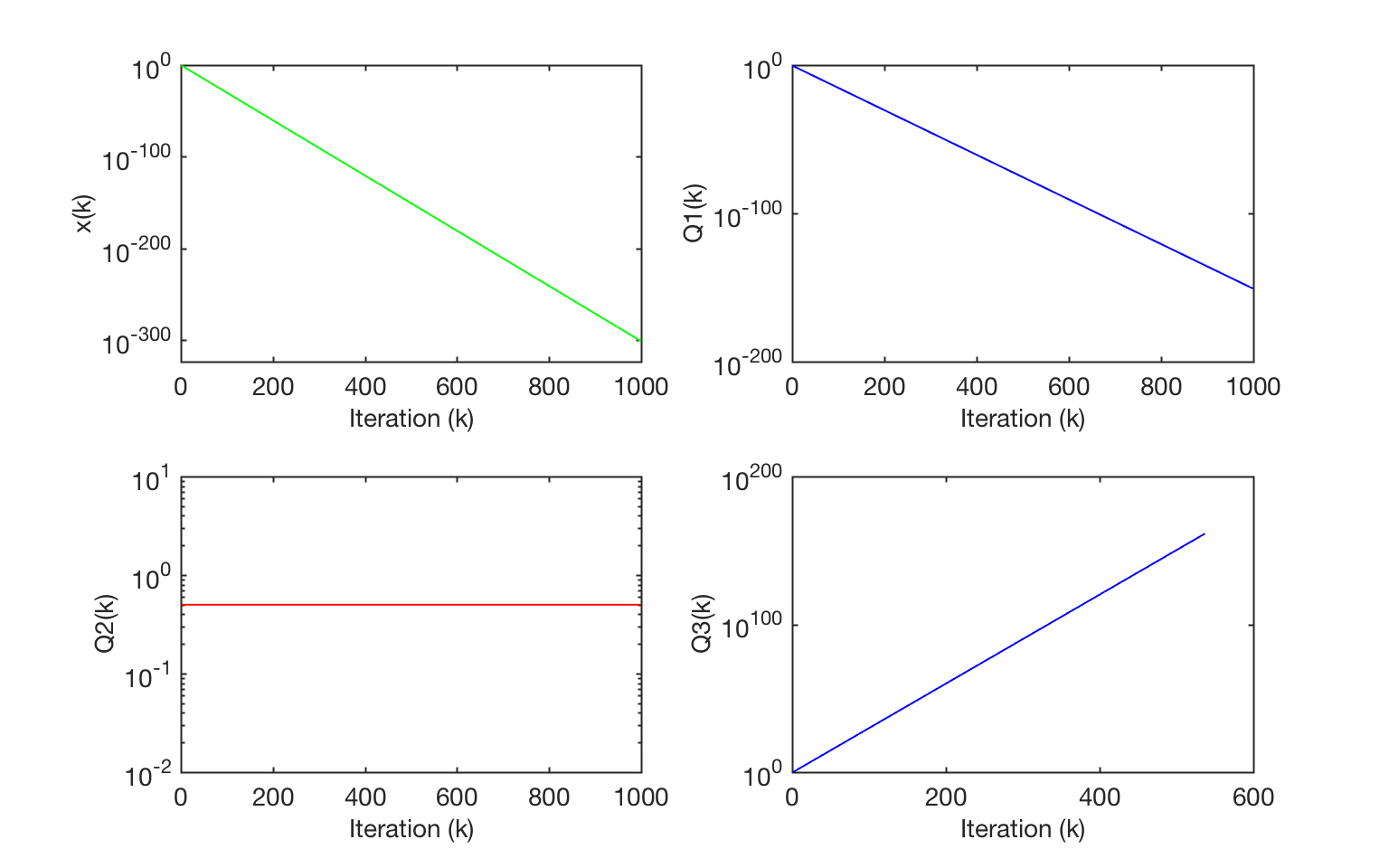
Linear\_convergence = figure;set(gcf,'Unit','Normalized','Position',[0 0.4 0.6 0.6])

subplot(2,2,1);semilogy([1:niter],x,'-g');xlabel('Iteration (k)');ylabel('x(k)')

subplot(2,2,2);semilogy([1:niter-1],Q1(1:niter-1),'-b');xlabel('Iteration (k)');ylabel('Q1(k)')

subplot(2,2,3);semilogy([1:niter-1],Q2(1:niter-1),'-r');xlabel('Iteration (k)');ylabel('Q2(k)')

subplot(2,2,4);semilogy([1:niter-1],Q3(1:niter-1),'-b');xlabel('Iteration (k)');ylabel('Q3(k)')



Finally, let . This is a case of *quadratic convergence*:

In this case, the rate of convergence is *r = 1* and the order of convergence is *p = 2*.

clear x Q1 Q2 Q3

niter = 9;

for k = 1:niter

x(k) = 1/(2^(2^k));

end

Q1 = abs(x(2:end))./(abs(x(1:end-1)).^1)

Q2 = abs(x(2:end))./(abs(x(1:end-1)).^2)

Q3 = abs(x(2:end))./(abs(x(1:end-1)).^3)

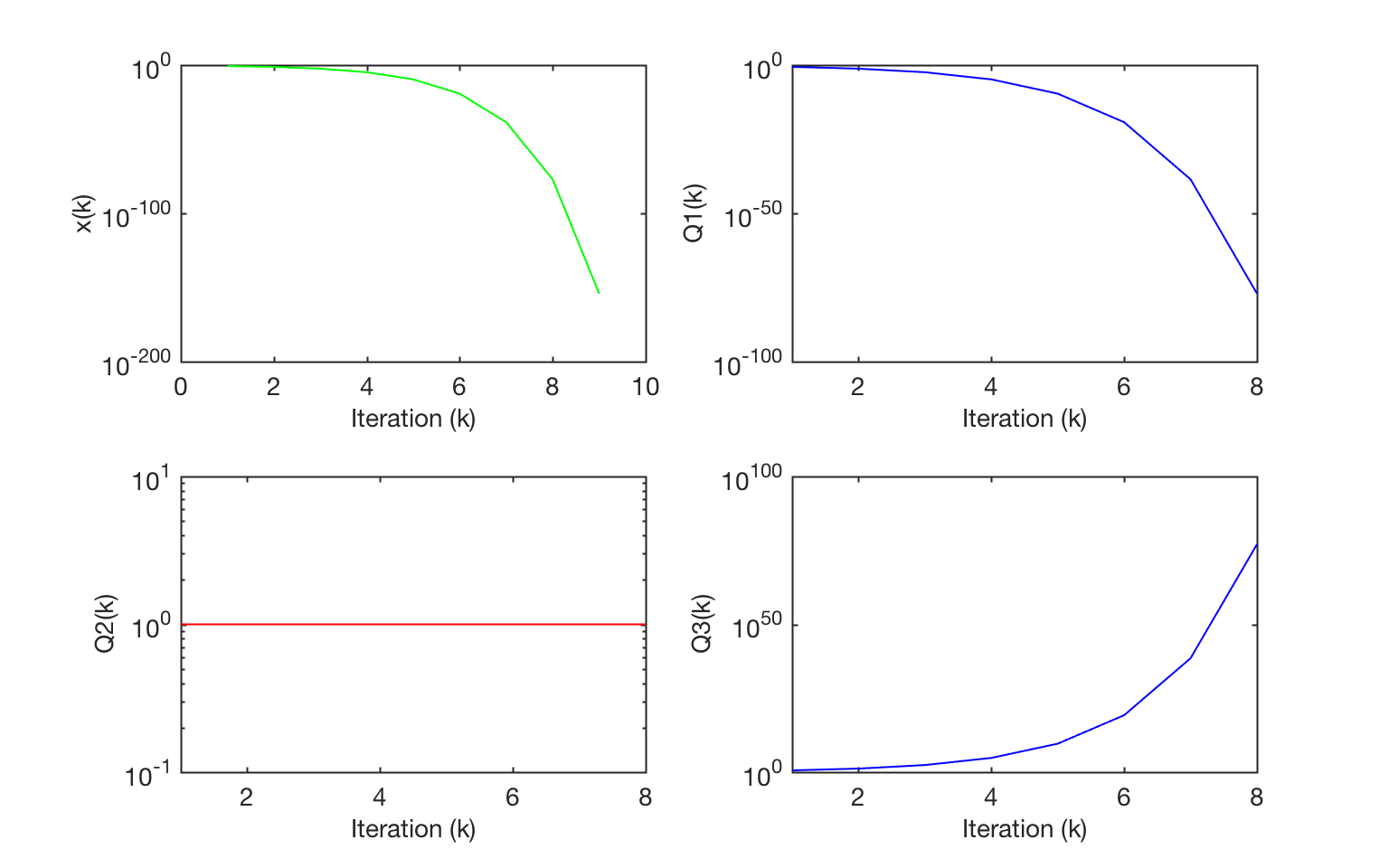
Quadratic\_convergence = figure;set(gcf,'Unit','Normalized','Position',[0 0.4 0.6 0.6])

subplot(2,2,1);semilogy([1:niter],x,'-g');xlabel('Iteration (k)');ylabel('x(k)')

subplot(2,2,2);semilogy([1:niter-1],Q1(1:niter-1),'-b');xlabel('Iteration (k)');ylabel('Q1(k)')

subplot(2,2,3);semilogy([1:niter-1],Q2(1:niter-1),'-r');xlabel('Iteration (k)');ylabel('Q2(k)')

subplot(2,2,4);semilogy([1:niter-1],Q3(1:niter-1),'-b');xlabel('Iteration (k)');ylabel('Q3(k)')



From these examples we see that there is a unique exponent , the *order of convergence*, so that:

Smaller values and larger values assure faster convergence. Thus, there is a lot to gain to find the Newton direction ( versus the Steepest descent direction (.

In practice an approximate estimate of the convergence order is derived as the limit of the following formula:

num = (x(4:end)-x(3:end-1))./(x(3:end-1)-x(2:end-2));

den = (x(3:end-1)-x(2:end-2))./(x(2:end-2)-x(1:end-3));

p = log(abs(num))./log(abs(den))

**BFGS (Broyden-Fletcher-Goldfarb-Shanno method).** We have seen howat each iteration the methods that use gradient information build a model based on the Taylor expansion of the objective around the current point **:**

and take the *line search* directionas the *steepest descent* (*SD*) vector:

or as the *Newton vector:*

to find the next iterate . If and are known, and is positive definite, one direct way to compute the Newton direction is to use a CG iteration to solve the system of linear equations:

*Newton-type* methods (see below) calculate directly at each iteration. *Quasi-Newton* methods avoid this by building an *approximation* of at each iteration using an appropriate updating technique. At each iteration, also the model is updated:

A large number of Hessian updating methods have been developed. However, the formula of *Broyden, Fletcher, Goldfarb, and Shanno* (BFGS) is the most effective for general use.

The updating formula given by BFGS, which assures that converges to in the search direction is:

where:

The gradient information is usually derived by partial derivatives using a numerical differentiation method *via* *finite differences*. Positive definiteness of is accomplished by ensuring that is initialized as positive definite (i.e., as the identity matrix, ***I***) and that the term is always positive. This term is a product of the line search step length parameter and a combination of the search direction with past and present gradient evaluations,

The condition that is positive is obtained by performing a sufficiently accurate line search. This is because the search direction, , is a descent direction, so that and the term are always positive. The term is negative, but can be made as small as required by increasing the accuracy of the line search, which makes the gradient at the iteration smaller than that at the iteration.

Since at each iteration we need to recalculate the Newton direction to carry out the line search, the BFGS method introduces a new Hessian defined as:

based on which the update formula becomes:

The complete flow-chart for the BFGS methods is the following:

Start with point

**⇓**

Use the *steepest descent* as 1st step

**⇓**

**⇓**

while

**⇓**

Do a Line Search to find

**⇓**

**⇓**

**⇓**

which is implemented in the function *bfgs*:

function [x1,f1,niter] = bfgs(f,x0,h,tolgrad,toldiff,tolline,nder,max\_b\_iter,max\_s\_iter)

% f = function to be minimized

% x = starting point as a column vector.

% h = initial search increment in the line search.

% nder = type of derivative: [1] 1st forward [2] 2nd forward [3] central

% tolgrad = error tolerance on the gradient.

% toldiff = error tolerance on the var diff.

% tolline = error tolerance on the minimum of f(x) in the line search.

% max\_b\_iter = maximum number of iterations in the bracketing phase.

% max\_s\_iter = maximum number of iterations in the sectioning phase.

if nargin < 9; max\_s\_iter = 10000; end

if nargin < 8; max\_b\_iter = 10000; end

if nargin < 7; nder = 3; end

if nargin < 6; tolgrad = 1.0e-8; end

if nargin < 5; toldiff = 1.0e-6; end

if nargin < 4; tolgrad = 1.0e-4; end

if nargin < 3; h = 0.01; end

n = size(x0,1);

I = eye(n);

H0 = I;

[~,g0] = gradx(f,x0,nder);

niter = 0;

s = toldiff;

while norm(s) >= toldiff

        if (g0'\*g0) <= tolgrad

            return

        end

        niter = niter + 1;

        p = -H0\*g0;

        % Line search.

        [s,~] = line\_search(@(s) f(x0+s\*p),0.0,-Inf,Inf,h,tolline,max\_b\_iter,max\_s\_iter);

        x1 = x0+s\*p;

        s = x1 - x0;

        [f1,g1] = gradx(f,x1,nder);

        q = g1 - g0;

        rho = 1/(q'\*s);

        H1 = (I - rho\*s\*q')\*H0\*(I-rho\*q\*s') + rho\*(s\*s');

        x0 = x1;g0 = g1;H0 = H1;

end

end

func = @(x) 10\*x(1)^2 + 3\*x(2)^2 -10\*x(1)\*x(2) + 2\*x(1)

x = [1 1]'

[xMin1,fMin1,nCyc1] = bfgs(func,x,0.0001,1.0e-4,1.0e-8,1.0e-8,1,1.0e4,1.0e4)

xMin1 = [-0.6000 -1.0000]

fMin1 = -0.6000

nCyc1 = 3

func = @(x) sin(x(1)) + x(2)^2 + log(x(3)) -7

x = [5 7 9]'

[xMin2,fMin2,nCyc2] = bfgs(func,x,0.00001,1.0e-4,1.0e-8,1.0e-8,2,1.0e4,1.0e4)

xMin2 = [5.0250 -0.0015 0.0000]

fMin2 = -46.8859

nCyc2 = 4

An alternative to the *quasi-Newton* type BFGS method is the *Newton* type Trust Region method. As before, the model around isbased on the Taylor expansion of the objective around the current point **:**

truncated to 2nd order elements, where is the *Newton vector:*

which, as we have seen, can be used to find the next iterate in the search of the minimum of. However, the *Newton vector* is in itself the **unconstrained** *full minimization* (*FS*) *step*, , of the quadratic model . So, we can equivalently write:

Likewise, the *steepest descent* (*SD*) *step*, , of calculated as:

leads to the **unconstrained** minimum of the quadratic model along the steepest descent direction. Along with the model of we also define a region in which we trust the model to be a good representation of the objective . Usually the initial trust region is defined by its radius :

Notice that if:

we must find the **constrained** solution to the *trust region subproblem*:

Notice how the *trust region* is implemented by constraining . The optimizer of the model in this trust region becomes the next iterate .

We will not discuss here how this constrained minimum is found (see CHAPTERS 7,18 for other examples of constrained minimization), except for mentioning that in the so called 2D-*subspace* method, implemented in the MATLAB function *fminunc*, the minimization is staged to include all the vectors in the plane spanned by the *full minimization step* and the *steepest descent step*:

s.t.

Once a trial step is calculated a simple algorithm based on the ratio between actual and predicted reduction in function value is used to decide whether the step is accepted or not:

If (i.e., ) we shrink the size of the trust region; if it is safe to expand the trust region; in all other cases we leave unchanged.

Clearly, *Newton-type* methods, including the *trust region* methods, must calculate directly at each iteration. If isnot positive definite (e.g., indefinite), it is possible to condition it by adding proper multiples (e.g., the negative of the largest negative eigenvalue) to the identity matrix so that the more general expression for becomes:

There are also other forms of conditioning based on just flipping the sign of the smallest and negative eigenvalues. If is positive semidefinite, it is also possible to use the *pseudoinverse* of .

Of course, if is positive definite, the *Trust-region subproblem* can be solved using with a CG iteration that solves the system of linear equations:

stopping the iteration when . Ultimately, when we get close to the minimum of , the trust-region constraint becomes inactive, as the model becomes a good approximation of the objective and the radius of the trust region grows.

Both the BFGS method and the Trust-region/2D-subspace method are implemented with a large range of options inside MATLAB *fminunc* function for unconstrained minimization:

func = @(x) sin(x(1)) + x(2)^2 + log(x(3)) -7

x = [5 7 9]

*fminunc* with internal calculation of gradients and Hessian:

options = optimoptions(@fminunc,'Algorithm','quasi-Newton',...

'MaxFunEvals',5000,'MaxIter',5000,'GradObj','off',...

'FinDiffType','central','DerivativeCheck','on',...

'Display','final','TolFun',1e-9,'TolX',1e-9);

[xMin,fMin] = fminunc(func,x,options)

*fminunc* with external function for gradients calculation, but internal Hessian calculation:

fg = @(x) gradx(func,x,3);

options = optimoptions(@fminunc,'Algorithm','quasi-Newton',...

'MaxFunEvals',5000,'MaxIter',5000,'GradObj','on',...

'FinDiffType','central','DerivativeCheck','on',...

'Display','final','TolFun',1e-9,'TolX',1e-9);

[xMin,fMin] = fminunc(fg,x,options)

*fminunc* with external function for gradients and internal Hessian calculation, but initial external Hessian provided as the identity matrix:

fg = @(x) gradx(func,x,3);

options = optimoptions(@fminunc,'Algorithm','quasi-Newton',...

'MaxFunEvals',5000,'MaxIter',5000,'GradObj','on',...

'FinDiffType','central','DerivativeCheck','on',...

'InitialHessMatrix',[1 1 1],'InitialHessType','user-supplied',...

'Display','final','TolFun',1e-9,'TolX',1e-9);

[xMin,fMin] = fminunc(fg,x,options)

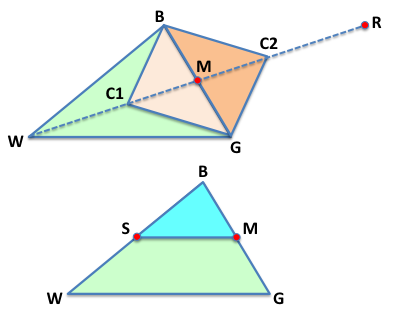
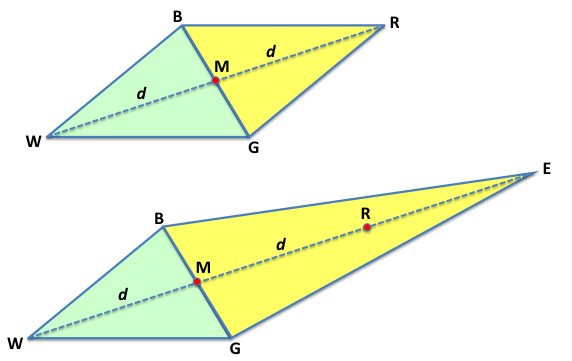
**Downhill Simplex.**

When the *n*-dimensional surface representing the function is extremely irregular, minimizers based on conjugate directions/gradients or higher order derivatives can be easily trapped in small local minima. In these cases, or in other cases in which the function may have discontinuities, it is preferable to adopt a less efficient (=slower), but more robust method that does not use derivatives and can more easily hop over small bumps or gaps in the surface. The most popular of these methods is the *Nelder-Mead* method, or *Downhill Simplex*. In *n-*dimensional space a *simplex* is a figure of *n+1* vertices connected by straight lines and bounded by polygonal faces. If *n=*2, a simplex is a *triangle*, if *n*=3, the simplex is a tetrahedron.

The *downhill simplex* algorithm can be better described by using as an example an *n*=2 dimensional function , for which the simplex is a triangle. The method compares *function values* at 3 vertices: the vertex at which is the largest is discarded and replaced with a new vertex. A new triangle is formed and the search continues generating a sequence of triangles, possibly of different shapes, for which the function values at the vertices become smaller and smaller.

The algorithm first makes a simplex around the initial guess ***x0*** by adding 5% of each component ***x0i*** to ***x0***, and using these *n* vectors as elements of the simplex. For example, if ***x*** = (1,1) the starting simplex has vertex coordinates ***v1*** = (1.05, 1), ***v2*** = (1, 1.05), ***v3*** = (1.05, 1.05). Then, the algorithm modifies the simplex repeatedly according to the following procedure. We use here the convention of referring to the three vertexes of the simplex as ***B*** (for best), ***G*** (for good), and ***W*** (for worst) based on the function values at the three points.

*Step1*: midpoint of the Good side. The midpoint ***M*** of the line segment joining ***B*** and ***G*** is found by averaging their coordinates. Its distance from ***W*** is *d*:



*Step 2*: reflection using the point ***R***. The function decreases along the edge from ***W*** to ***B***, and from ***W*** to ***G***, thus it is possible that it might take even smaller values on some points beyond the ***B-G*** edge and away from ***W***. For this reason we choose a point ***R*** obtained by *reflecting* the triangle through the ***B-G*** edge:

*Step 3*: expansion using the point ***E***. If the function value at ***R*** is smaller than at ***W***, then it is possible that moving even further than ***R*** would decrease the function further. Thus, we move by an additional distance *d* along the line from ***M*** to ***R*** forming the triangle ***BGE***:

If the function value at ***E*** is smaller than at ***R***, than we have found a better vertex than ***R***.

*Step 4*: contraction using the point ***C***. If ***R*** is not better than ***W***, we have to find another point. ***M*** might be better than ***R***, but we can't choose a point on an edge because we need to form a triangle. We can choose two midpoints ***C1*** (on the line segment ***WM***) and ***C2*** (on the line segment ***MR***): the point with the smaller value becomes ***C*** in the new triangle ***BGC***.

*Step 5*: shrinkage toward ***B***. If ***C*** is not better than ***W***, than the points ***G*** and ***W*** must shrink toward ***B***: ***G*** is replaced by ***M***, and ***W*** is replaced by ***S***, the midpoint of the ***W-B*** edge.

The process continues and generates a sequence of triangles that converges down on the solution point as soon as a value of the function at one of the vertexes becomes smaller than a given tolerance value.

A Table for the logical decisions made at each step is shown below:

|  |  |
| --- | --- |
| If f(***R***)<f(***G***) THEN  carry out Case 1(reflect or extend)  ELSE  carry out Case 2 (contract or shrink) | |
| BEGIN Case 1 | BEGIN Case 2 |
| IF f(***B***) < f(***R***) THEN | IF f(***R***) <f(***W***) THEN |
| replace ***W*** with ***R*** | replace ***W*** with ***R*** |
| ELSE | ELSE |
| Compute ***E*** and f(***E***) | compute ***C1*** = (***W***+***M***)/2 and ***C2*** = (***M***+***R***)/2 |
|  | compute f(***C***) = min(f(***C1***),f(***C2***)) |
| IF f(***E***)<f(***B***) THEN | IF f(***C***)<f(***W***) THEN |
| replace ***W*** with ***E*** | replace ***W*** with ***C*** |
| ELSE | ELSE |
| replace ***W*** with ***R*** | compute ***S*** and f(***S***) |
|  | replace ***W*** with ***S*** |
|  | replace ***G*** with ***M*** |
| ENDIF | ENDIF |
| ENDIF | ENDIF |
| END | END |

A MATLAB implementation of the *downhill simplex* algorithm is provided by the function *fminsearch*:

func = @(x) sin(x(1)) + x(2)^2 + log(x(3)) -7

x = [5 7 9]

options = optimset('MaxFunEvals',5000,'MaxIter',5000,'Display','final','TolFun',1e-8,'TolX',1e-8);

[xMin,fMin] = fminsearch(func,x,options)

Restarting the minimization from the result obtained in the previous run allows the method to jump over small irregularity in the surface:

for i = 1:100

[xMin,fMin,exitflag,output] = fminsearch(func,xMin,options)

end